集成学习

1.集成学习是什么

1.1 基本思想

（1）“弱者的联盟”

集成学习(Ensemble learning)是机器学习中的一种思想，通过构建并结合多个个体学习器（Individual learner）形成一个精度更高的机器学习模型。这些个体学习器也是机器学习算法，可以是朴素贝叶斯、决策树、支持向量机和神经网络等。集成学习示意图如**图1**所示。

传统机器学习算法 (例如：决策树，逻辑回归等) 的目标都是寻找一个最优分类器尽可能的将训练数据分开。集成学习 算法的基本思想就是将多个弱分类器组合，从而实现一个预测效果更好的集成分类器。集成算法可以说从一方面验证了中国的一句老话：三个臭皮匠，赛过诸葛亮。

> **Note**
>
> 弱分类器（Weak learner）是指在特定学习任务上性能略高于随机猜测的学习器。

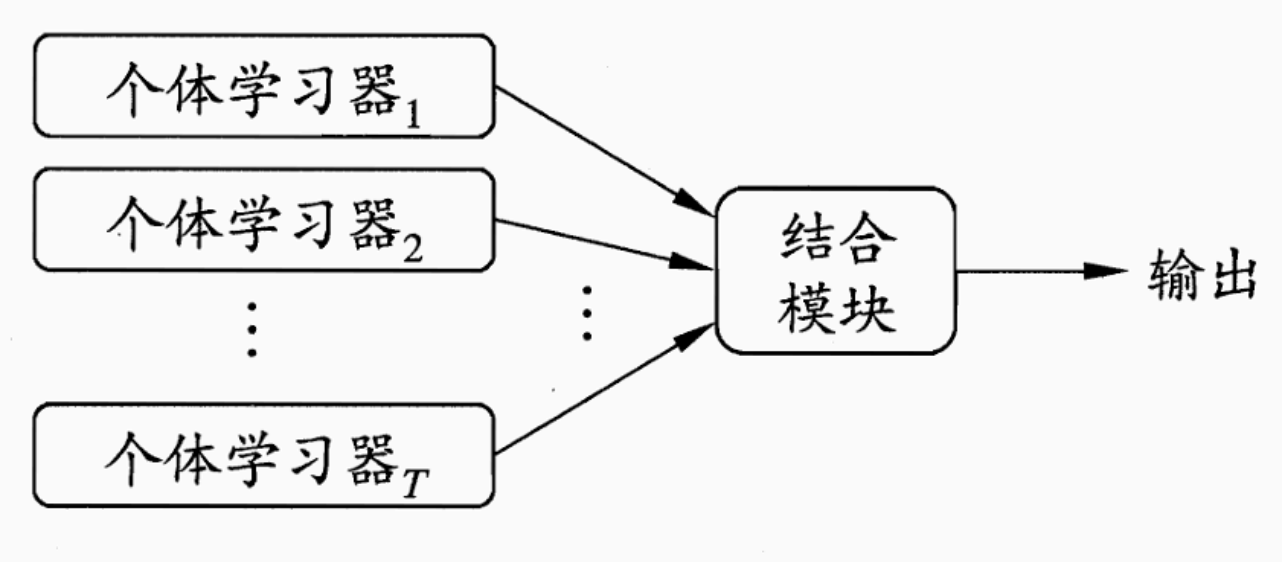


图1.集成学习示意图

（2）“多样性红利”：模型间的差异性比单个模型的精度更重要，这与人类社会团队协作的规律惊人相似。

图2.不同集成结果示例

1.2 集成学习的类型

方法	核心策略	通俗理解	典型算法
Bagging	并行训练+投票/平均	民主决策	随机森林
Boosting	串行训练+错误修正	知错能改	AdaBoost,XGBoost
Stacking	分层训练+元模型融合	专家委员会	Blending

1.3 集成学习的结合策略

1. 写在前面：为什么结合策略是集成学习的灵魂？

集成学习的核心不是“模型越多越好”，而是“如何让模型间的协作产生超越个体的智慧”。真正决定集成效果上限的，往往是**基学习器的结合策略**（Combination Strategy）。

> **Note**
>
> **我的思考：**
>
> - 如果把基模型比作“专家”，结合策略就是“专家委员会”的议事规则；
> - 好的策略能抑制噪声、放大有效信息，甚至让弱模型通过协作达到强模型的效果；
> - 结合策略的设计本质是**信息融合的数学建模**，背后隐含对数据分布、模型能力的先验假设。

2. 经典策略

(1)投票法（Voting）

- 硬投票（Hard Voting）：平等对待每个模型，易受“多数暴政”影响（噪声模型可能主导结果）

$$\hat{y} = \text{argmax}_{c \in C} \sum_{i=1}^{T} \mathbb{I}(h_i(x) = c)$$

其中：

C：类别集合；II(·)：指示函数（预测为类别 cc 时取1，否则取0）。

特点:直接统计类别票数，多数决制；可能受“多数噪声模型”干扰（若多个弱模型预测错误）。

- 软投票（Soft Voting）：引入概率权重，但对置信度的校准敏感（模型输出概率未必可靠）。

$$\hat{y} = \text{argmax}_{c \in C} \frac{1}{T} \sum_{i=1}^{T} P_i(c|x)$$

其中：

- $P_i(c|x)$表示第i个模型对样本x属于类别c的预测概率。

特点：要求基模型能输出概率（如逻辑回归、带概率校准的SVM）；对模型校准敏感，若概率未校准可能效果下降。

- 加权投票（Weighted Voting）

以软投票为例：

$$\hat{y} = \text{argmax}_{c \in C} \sum_{i=1}^{T} w_i P_i(c|x)$$

- 权重wi可基于模型性能或领域知识设定（如AUC值高的模型权重更大）。

> **Note**
>
> - 是否所有模型的“投票权”应该平等？

- 如何量化模型在不同样本区域的置信度？
- **改进思路**：动态权重分配（如基于样本局部密度的加权投票）。

(2)平均法（Averaging）

- 简单平均（Simple Averaging）

$$\hat{y} = \frac{1}{T}\sum_{i=1}^{T} h_i(x)$$

其中：*T*：基模型数量；h_i(x)：第 *i* 个模型对样本 *x* 的预测值；y^：最终预测结果。

特点：所有模型权重相等，假设模型误差服从独立同分布；对异常值敏感（可通过截断平均改进）。

- 加权平均（Weighted Averaging）

$$\hat{y} = \sum_{i=1}^{T} w_i h_i(x), \quad \text{其中} \sum_{i=1}^{T} w_i = 1$$

wi：第*i*个模型的权重，通常根据模型性能（如验证集准确率）动态分配。

特点：高性能模型获得更高权重；需注意权重分配的合理性（避免过拟合验证集）。

> **Tip**
>
> - 算术平均假设误差服从高斯分布，但现实任务中误差可能呈现偏态或重尾分布。
> - **案例**：在金融风险预测中，少数极端值的预测误差可能对简单平均产生灾难性影响。
> - **解决方案**：
>
>   **截断平均（Trimmed Mean）**：去掉最高/最低的预测值；
>
>   **分位数融合（Quantile Blending）**：直接集成不同分位数的预测结果。

(3) 学习法

- 传统Stacking用基模型的输出训练元模型，但可能引入过拟合风险（尤其在基模型高度相关时）。
- **我的实验发现**：

- 使用**低复杂度的元模型**（如线性回归）反而比深度网络更稳定；
- 对基模型输出做**特征工程**（如加入原始特征、交互项）比直接拼接更有效；
- **对抗验证技巧**：通过检测元模型是否过拟合基模型的噪声来调整训练策略。

---

3. 关键对比与选择建议

方法	适用场景	优点	缺点
简单平均	模型性能相近的回归任务	计算简单，抗过拟合	对异常值和低质量模型敏感
加权平均	模型性能差异显著的回归任务	灵活利用模型差异性	需额外计算权重，可能过拟合验证集
硬投票	类别标签明确的分类任务	无需概率输出，实现简单	忽略模型置信度，易受多数噪声影响
软投票	模型输出可靠概率的分类任务	利用概率信息，结果更平滑	依赖概率校准，计算复杂度略高

---

2. 随机森林

2.1算法思想

1.基本概念

随机森林=决策树+Bagging+随机特征选择

核心思想："三个臭皮匠顶个诸葛亮"，通过构建多个弱学习器（决策树）的群体智慧提升预测效果。

（1）双重随机性数学表达

- Bootstrap抽样：从原始数据集DD中有放回抽取n个样本，生成子集Dk

  单个样本不被选中的概率：

$$\lim_{n\to\infty}(1-\frac{1}{n})^n=\frac{1}{e}\approx 36.8\%$$

- **特征随机选择**：每次节点分裂时，从p个特征中随机选取m个候选特征（分类问题$m=\lfloor p \rfloor$，回归问题$m=\lfloor p/3 \rfloor$）

（2）群体决策模式

- 分类任务：多数投票制（消除单棵树的偏见）
- 回归任务：均值计算（平滑极端预测值）

（3）自验证特性

- 未被抽中的36.8%数据（OOB样本）自动成为验证集
- 无需额外划分验证集即可评估泛化能力

2.决策树生长细节

```
class Node:
    def split(self, X, y):
        best_gini = float('inf')
        # 遍历所有候选特征
        for feature in random_features:
            # 遍历所有分割点
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                left_idx = X[:, feature] <= threshold
                gini = self._calc_gini(y[left_idx], y[~left_idx])
                if gini < best_gini:
                    best_gini = gini
                    self.split_rule = (feature, threshold)
```

Gini不纯度计算：

$$\text{Gini}(t) = 1 - \sum_{i=1}^{c} [p(i|t)]^2$$

其中c为类别数，$p(i \mid t)$为节点t中第i类的比例。

2.2算法参数

参数	模型复杂度	训练速度	过拟合风险	建议调参范围
`n_estimators`	↑	↓	↓	100-500（优先调整）
`max_depth`	↑↑	↓	↑↑	3-15
`min_samples_leaf`	↓	↑	↓↓	1-20
`max_features`	↓	↑	↓	sqrt/auto/log2

【实践】随机森林进行银行危机预测并将其与决策树进行比较

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, recall_score,
precision_score, f1_score
from matplotlib import pyplot as plt
```

```
# 1.数据预处理
data = pd.read_csv('../data/crisis.csv', encoding='ansi')  # 读入数据
data = data.drop(['国家'], axis=1)  # 删除特征
data['银行危机'] = LabelEncoder().fit_transform(data['银行危机']) # 对
“银行危机”进行编码
X_train, X_test, y_train, y_test = train_test_split(data.drop('银行
危机', axis=1), data['银行危机'], random_state=666)  # 划分数据集

# 2.模型构建和训练
dt_clf = DecisionTreeClassifier()  # 构建决策树模型
dt_clf.fit(X_train, y_train)  # 训练决策树模型
y_pred_dt = dt_clf.predict(X_test)  # 用决策树模型进行预测
rf_clf = RandomForestClassifier()  # 构建随机森林模型
rf_clf.fit(X_train, y_train)  # 训练随机森林模型
y_pred_rf = rf_clf.predict(X_test)  # 用随机森林模型进行预测
```

```
# 3.打印模型评价
def print_model_evaluation(model_name, y_pred):
    print(f'***** {model_name} *****')
```

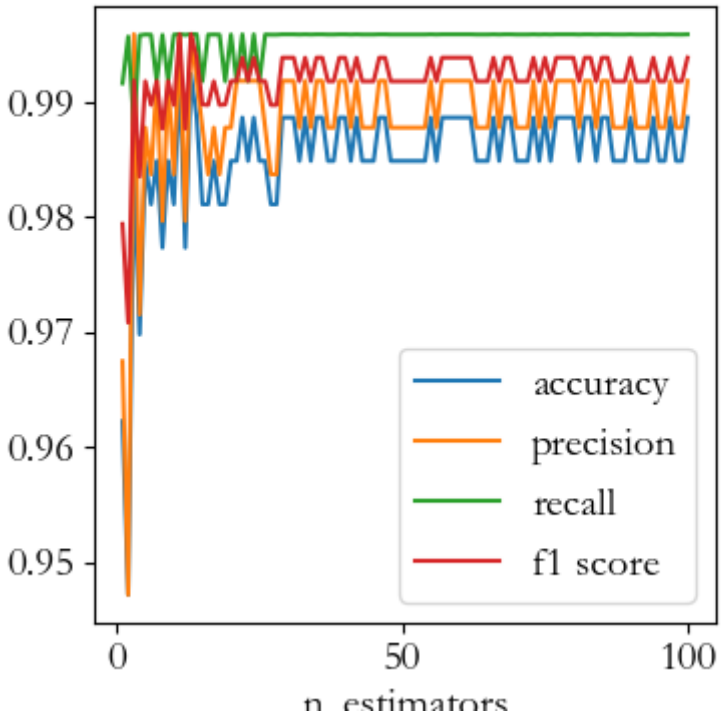
```
    print('accuracy', accuracy_score(y_pred, y_test))
    print('precision', precision_score(y_pred, y_test))
    print('recall', recall_score(y_pred, y_test))
    print('f1 score', f1_score(y_pred, y_test))
print_model_evaluation('决策树', y_pred_dt)
print_model_evaluation('随机森林', y_pred_rf)

# 4.绘制n_estimators参数对预测性能的影响曲线
accuracy_list, precision_list, recall_list, f1_list = [], [], [],
[]
for n in range(1,101):
    rf_clf = RandomForestClassifier(n_estimators=n)  # 构建随机森林模
型
    rf_clf.fit(X_train, y_train)
    y_pred_rf = rf_clf.predict(X_test)
    accuracy_list.append(accuracy_score(y_pred_rf, y_test))
    precision_list.append(precision_score(y_pred_rf, y_test))
    recall_list.append(recall_score(y_pred_rf, y_test))
    f1_list.append(f1_score(y_pred_rf, y_test))

# 5.绘制性能曲线
plt.rcParams['font.sans-serif']=['STSong']
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(4, 4))
plt.plot(list(range(1, 101)), accuracy_list, label='accuracy')
plt.plot(list(range(1, 101)), precision_list, label='precision')
plt.plot(list(range(1, 101)), recall_list, label='recall')
plt.plot(list(range(1, 101)), f1_list, label='f1 score')
plt.xlabel('n_estimators')
plt.ylabel('评价指标')
plt.legend()
plt.show()
```

结果：

```
***** 决策树 *****
accuracy 0.9660377358490566
precision 0.967479674796748
recall 0.99581589958159
f1 score 0.9814432989690722
***** 随机森林 *****
accuracy 0.9849056603773585
precision 0.9878048780487805
recall 0.9959016393442623
f1 score 0.9918367346938776
```



算法特点总结

优势	局限
✅ 天然抗过拟合（双重随机性）	❌ 无法捕捉特征间交互效应
✅ 自动处理缺失值（通过替代分裂）	❌ 高维稀疏数据表现较差
✅ 输出特征重要性（基于分裂贡献度）	❌ 内存消耗大（需存储所有树结构）

3. AdaBoost

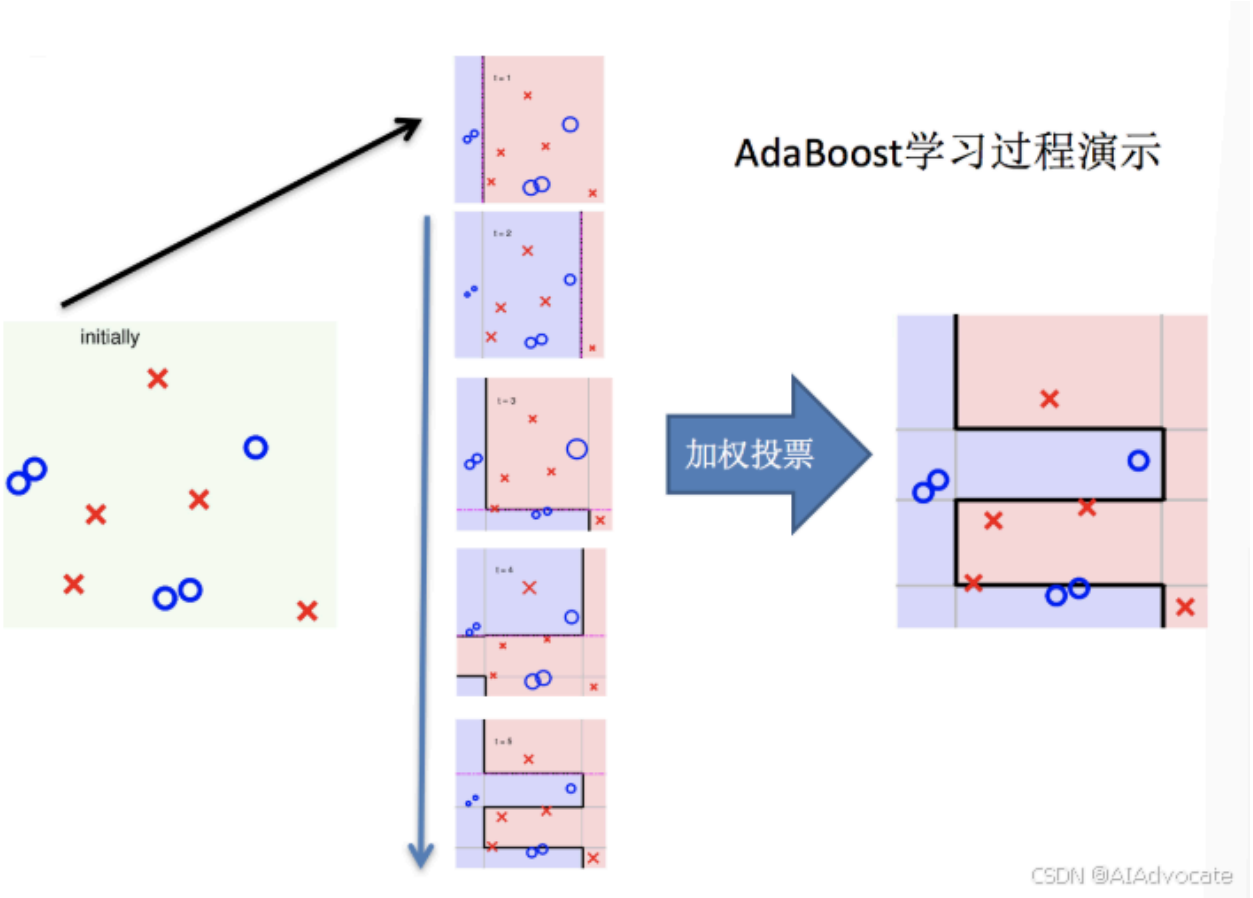
3.1 AdaBoost算法的基本思想

AdaBoost（Adaptive Boosting）是一种提升（Boosting）方法，它通过组合多个弱分类器（weak classifiers）来构建一个强分类器（strong classifier）。

其核心思想是：

- 训练多个弱分类器，每个分类器在上一个分类器的基础上进行优化，使得新分类器关注先前分类错误的样本。
- 通过加权投票或加权求和的方式组合这些弱分类器，使得最终的分类结果更加准确。
- 训练过程中，样本的权重会动态调整，错误分类的样本权重增大，使得后续分类器更加关注这些易错样本。

AdaBoost的优势在于能够显著提升分类性能，同时具有较强的抗过拟合能力。但其缺点是对噪声数据较为敏感，因为错误分类的样本权重增加可能导致模型过度拟合噪声数据。



> *https://blog.csdn.net/2201_75415080/article/details/143103726*

3.2 AdaBoost的构建过程

输入:

- 训练样本集：D={(x1,y1),(x2,y2),...,(xn,yn)}，其中xi 为输入特征，yi为类别标签（+1 或 -1）。
- 设定弱分类器（如决策树桩）作为基学习器。
- 初始化样本权重分布：wi = 1/n 。

迭代训练（T轮迭代）:

1. 训练弱分类器
    - 在当前样本权重分布下，训练一个弱分类器 h_t(x)，使其尽可能减少加权分类误差。
2. 计算分类误差
    - 计算弱分类器 h_t(x) 在加权样本集上的错误率：

$$\varepsilon_t = \sum_{i=1}^{n} w_i I(h_t(x_i) \neq y_i)$$

    - 如果 错误率 > 0.5，说明该分类器比随机猜测还差，直接终止训练。
3. 计算弱分类器的权重
    - 根据分类误差计算该分类器的权重：

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$

    - 该权重表示该分类器在最终分类器中的贡献程度，错误率越小，则权重越大。
4. 更新样本权重
    - 重新调整样本权重，使错误分类的样本获得更大的权重，正确分类的样本获得较小的权重：

$$w_i^{(t+1)} = w_i^{(t)} e^{-\alpha_t y_i h_t(x_i)}$$

    - 归一化权重，使其总和为1。
5. 重复以上过程，直到达到预设的弱分类器个数 T 或者分类误差降至某个阈值。

最终强分类器

训练完成后，最终的分类器是多个弱分类器的加权投票：

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

【具体例子】

数据集，假设有 5 个样本：

样本	特征	真实标签
1	1.0	-1
2	2.0	-1
3	3.0	+1
4	4.0	+1
5	5.0	+1

训练第一轮弱分类器

使用单层决策树作为弱分类器，例如：

- **分类规则**：如果 ，预测 -1，否则预测 +1。
- **分类结果**：
    - 样本 1, 2: 预测正确
    - 样本 3: 预测错误
    - 样本 4, 5: 预测正确
- **错误率**：

$$1/\varepsilon = 1/5 = 0.2$$

- **分类器权重**：

$$\alpha_1 = 0.5 \ln\left(\frac{1-0.2}{0.2}\right) = 0.69$$

- **更新样本权重**：
    - 对误分类的样本（样本 3），权重增加。
    - 其他样本权重降低。

如此迭代多轮，最终得到一个加权组合的强分类器。

构建最终强分类器

1. 最终的强分类器是所有弱分类器的加权和：

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

- 这里使用 **符号函数（sign function）** 进行最终分类。

3.3 思考与总结

AdaBoost的优势

- **权重的自适应性**：AdaBoost通过不断调整样本权重，使得后续的弱分类器更加关注难分类的样本，这种动态调整的机制是其核心亮点。
- **弱分类器的选择**：决策树桩（仅有一个分裂的树）是常见的弱分类器，因为它足够简单，但在复杂数据集上可能需要更复杂的基学习器。
- **计算复杂度**：AdaBoost本身计算开销不大，但随着迭代次数增加，计算复杂度会随之上升。

AdaBoost的局限性

- **对噪声数据敏感**：AdaBoost对噪声数据较敏感，容易过拟合异常值。在实践中，需要适当调整参数，如控制基学习器的复杂度，或者使用正则化方法来减轻过拟合。

【实践】AdaBoost和决策树相比较

```
import pandas as pd
from sklearn import ensemble
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from matplotlib import pyplot as plt
import numpy as np
import random
from itertools import accumulate
import warnings
warnings.filterwarnings("ignore")  # 忽略所有警告
```

```
# 1.数据预处理
df_acc_list, ad_acc_list = [], []
data = pd.read_csv("../data/horseColic.csv", names=[i for i in
range(28)])  # 导入数据
X, y = data.iloc[:,:-1], data.iloc[:,-1:]
model_num = 100
```

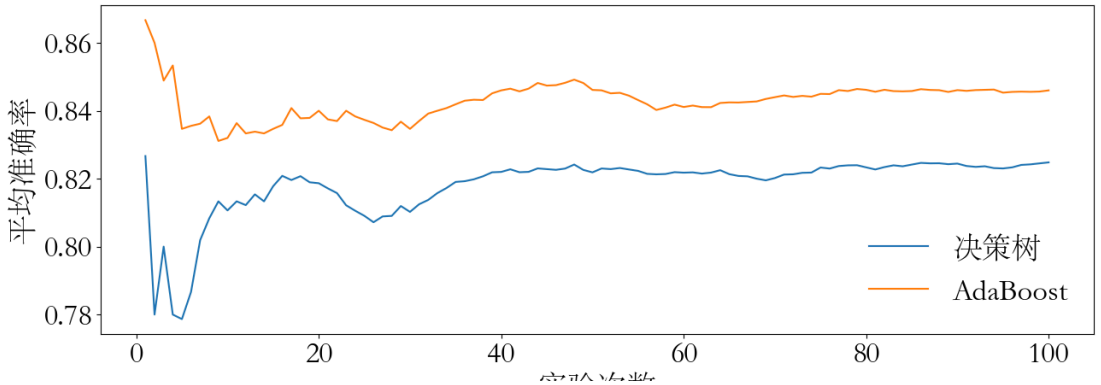
```
# 2.模型构建和训练
for seed in random.sample(range(1, model_num*2), model_num):
    # 随机划分为训练集和测试集
    X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state= seed)
    #构建决策树分类模型
    dtc = DecisionTreeClassifier(max_depth=7, min_samples_leaf=7)
    dtc.fit(X_train, y_train)
    df_acc_list.append(accuracy_score(y_test, dtc.predict(X_test)))
    #构建Adaboost分类模型
    clf =
ensemble.AdaBoostClassifier(base_estimator=DecisionTreeClassifier(m
ax_depth=7, min_samples_leaf=7), n_estimators=100,
algorithm='SAMME', learning_rate=0.95)
    clf.fit(X_train, y_train)
    ad_acc_list.append(accuracy_score(y_test, clf.predict(X_test)))
```

```
# 3.绘制性能曲线
plt.figure(figsize=(15,5))
font = {'family': 'serif', 'serif': ['STSong'], 'size':25}
plt.rc('font', **font)
plt.plot(list(range(1, model_num+1)),
np.array(list(accumulate(df_acc_list)))/np.array(range(1,
model_num+1)), label='决策树')
plt.plot(list(range(1, model_num+1)),
np.array(list(accumulate(ad_acc_list)))/np.array(range(1,
model_num+1)), label='AdaBoost')
plt.legend(frameon=False)
plt.xlabel('实验次数')
plt.ylabel('平均准确率')
plt.show()
```

```
# 4.打印模型评价
print('决策树平均准确率：',
round(sum(df_acc_list)/len(df_acc_list),4))
print('AdaBoost平均准确率：',
round(sum(ad_acc_list)/len(ad_acc_list),4))
```

结果：

决策树平均准确率： 0.8248
AdaBoost平均准确率： 0.846



4.梯度提升树（GBDT）

4.1 算法原理

Boosting框架：

- Boosting是一种迭代的加法模型（Additive Model），通过多个弱学习器（通常是决策树）逐步优化模型，使得整体模型误差不断减小。
- 与Bagging（如随机森林）不同，Boosting是串行训练的，每一棵新树都会考虑前一棵树的错误，专注于改进当前模型的不足。

残差学习：

- 每棵新树的目标是拟合前一轮的残差（即目标值与当前模型预测值的差）。
- 这样可以使新树的贡献方向与误差的梯度方向一致，从而逐步优化模型。

梯度优化：

- GBT的核心优化思想是**梯度下降**。在每一轮迭代中，我们计算损失函数（如平方误差、对数损失等）对预测值的梯度，并利用这个梯度信息来更新模型。
- 具体来说，新加入的树的输出是负梯度方向，以最小化损失函数。

弱学习器（Base Learner）：

- GBT通常使用CART（Classification and Regression Tree）回归树作为基学习器，每次训练一棵新的回归树，用来修正之前的误差。

学习率（Learning Rate）：

在更新模型时，每一棵树的贡献可以通过学习率（步长参数 η）进行缩放，以防止模型过拟合。

一般来说，较小的学习率可以提升泛化能力，但需要更多的树来达到同样的效果。

4.2 算法步骤

假设有一个训练数据集：D={(x1,y1),(x2,y2),...,(xn,yn)}

其中xi是特征向量，yi是真实值（回归任务）或类别标签（分类任务）。

算法流程：

1.初始化模型：

选择一个常数值作为初始预测值 F_0(x)，通常是目标变量的均值：

$$F_0(x) = \arg\min_c \sum_{i=1}^{n} L(y_i, c)$$

其中 L(y_i,c)是损失函数，如回归任务中常用平方误差损失：

$$L(y_i, c) = (y_i - c)^2$$

2.迭代训练决策树（Boosting 过程）： 对于每一轮迭代 m=1,2,...,M：

计算残差（梯度）：

$$r_{im} = -\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$$

这个残差实际上是目标值对当前模型输出的梯度，在回归任务中，若损失函数为平方误差：

$$r_{im} = y_i - F_{m-1}(x_i)$$

拟合一棵新的回归树：

用残差 r_im 作为新的目标值，训练一棵新的回归树 h_m(x)。

计算步长（Shrinkage）：

计算每个叶子节点的最佳分数 γ_m：

$$\gamma_m = \arg\min_{\gamma} \sum_{i \in \text{leaf}} L(y_i, F_{m-1}(x_i) + \gamma)$$

更新模型：

$$F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x)$$

其中 η是学习率，控制每棵树的贡献。

最终预测：经过 MMM 轮迭代后，得到最终模型：

$$\hat{y} = F_M(x)$$

【实例】

使用一个**回归问题**

样本X	真实值Y
1	5
2	10
3	15

目标是通过梯度提升树来学习 x 到 y 之间的映射关系。

步骤 1：初始化模型

最开始，用均值初始化模型：

$$F_0(x) = \frac{5 + 10 + 15}{3} = 10$$

因此，初始预测值：

样本X	真实值Y	初始预测值 F_0(X)
1	5	10
2	10	10
3	15	10

步骤 2：计算残差（负梯度）

残差（即负梯度）计算如下：

$$r_i = y_i - F_0(x_i)$$

样本 X	真实值 Y	预测值 F_0(X)	残差
1	5	10	-5
2	10	10	0
3	15	10	5

步骤 3：训练一棵回归树

现在，我们用残差作为目标值训练一棵小的回归树：

x=1 的残差是 -5

x=2 的残差是 0

x=3 的残差是 5

假设回归树学到了一个简单的规则：

$$h_1(x) = \begin{cases} -5, & x = 1 \\ 0, & x = 2 \\ 5, & x = 3 \end{cases}$$

步骤 4：更新模型

使用学习率 η=0.5来更新模型：

$$F_1(x) = F_0(x) + \eta \cdot h_1(x)$$

计算新的预测值：

样本 X	真实值 Y	新树的预测 H_1(X)	更新后的预测值 F1(X)=F0(X)+0.5·H1(X)
1	10	-5	10+0.5×(−5)=7.5

样本 X	真实值 Y	新树的预测 H_1(X)	更新后的预测值 F1(X)=F0(X)+0.5·H1(X)
2	10	0	10+0.5×0=10
3	10	5	10+0.5×5=12.5

步骤 5：计算新的残差

现在，我们基于新的预测值计算残差：

$$r_i^{(2)} = y_i - F_1(x_i)$$

样本 X	真实值 Y	预测值F1(X)	残差
1	5	7.5	-2.5
2	10	10	0
3	15	12.5	2.5

步骤 6：训练第二棵回归树

同样，我们训练一棵新的回归树来拟合新的残差：

$$h_2(x) = \begin{cases} -2.5, & x = 1 \\ 0, & x = 2 \\ 2.5, & x = 3 \end{cases}$$

步骤 7：更新模型

$$F_2(x) = F_1(x) + \eta \cdot h_2(x)$$

计算新的预测值：

样本 X	真实值 Y	新树的预测 H_2(X)	更新后的预测值 F2(X)=F1(X)+0.5·H2(X)
1	7.5	-2.5	7.5+0.5×(−2.5)=6.25
2	10	0	10+0.5×0=10
3	12.5	2.5	12.5+0.5×2.5=13.75

步骤 8：收敛

经过两轮迭代后，我们的模型预测值如下：

样本 X	真实值 Y	预测值F2(X)
1	5	6.25

样本 **X**	真实值 **Y**	预测值**F2(X)**
2	10	10
3	15	13.75

【个人思考与总结】

相比于随机森林，GBT更容易过拟合：

- 由于Boosting是串行训练的，每棵树都是在纠正前一棵树的错误，因此容易捕捉数据中的噪声，导致过拟合。
- 解决方案：
    - 采用较小的学习率（如 0.01~~0.1）+ 更多的树（如 100~~1000 棵）。
    - 采用正则化，如子采样（Subsample），L1/L2 正则化。

决策树的深度对GBT影响较大：

- 树太深：容易过拟合。
- 树太浅：单棵树的表达能力不足，学习效率下降。
- 一般经验：3~8层的决策树通常效果较好。

在数据量较大时，XGBoost更适合实际应用：

- 传统的梯度提升树（GBDT）计算复杂度较高，训练时间较长。
- XGBoost采用直方图加速、分裂优化、正则化等策略，使得训练效率大幅提高，适用于大规模数据。

学习率的选择是关键：

- **较小的学习率 + 较多的树**：稳定，泛化能力强，但计算开销大。
- **较大的学习率 + 较少的树**：训练快，但泛化能力可能下降。
- 经验法则：
    - 如果数据集较小（<10万），可以尝试较大的学习率（0.1~0.3）。
    - 如果数据集较大，建议使用 0.01~0.1 的学习率。

5.XGBoost

5.1 XGBoost 算法原理

XGBoost（eXtreme Gradient Boosting）是基于梯度提升决策树（GBDT, Gradient Boosting Decision Tree）的一种优化实现。相比于传统的 GBDT，XGBoost 具有更高效的计算速度和更强的模型表现力，广泛应用于分类、回归以及排名任务。

目标函数：

XGBoost 的目标函数由两部分组成：损失函数（Loss Function）和正则化项（Regularization Term）：

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

损失函数 l 衡量模型的预测值与真实值之间的误差，通常使用平方损失（回归）或对数损失（分类）。

正则化项 Ω(fk)控制树的复杂度，防止过拟合，定义如下：

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j} w_j^2$$

其中，T是树的叶子节点数，wj是叶子节点的权重，γ和 λ 是超参数，用于控制树的复杂度。

二阶泰勒展开：

为了高效优化目标函数，XGBoost 采用二阶泰勒展开近似损失函数：

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} [g_i f(x_i) + \frac{1}{2} h_i f^2(x_i)] + \Omega(f)$$

其中：

$$g_i = \frac{\partial l(y_i, \hat{y}_i)}{\partial \hat{y}_i} (一阶梯度)$$

$$h_i = \frac{\partial^2 l(y_i, \hat{y}_i)}{\partial \hat{y}_i^2} (二阶梯度)$$

这种优化方式相比于传统 GBDT 只使用一阶导数，能够更快地收敛，提高训练效率。

叶子节点的最佳分裂：

XGBoost 采用贪心策略进行最佳特征分裂。每个节点的增益计算公式如下：

$$Gain = \frac{1}{2}\left(\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right) - \gamma$$

- GL,HL 和 GR,HRG_R, H_RGR,HR 分别表示左子节点和右子节点的梯度和二阶梯度之和。
- λ 控制叶子权重的惩罚，γ 用于控制是否进行分裂（类似于剪枝）。

如果增益 Gain小于 0，则不进行分裂，以防止过拟合。

5.2 XGBoost 算法步骤

初始化预测值
设定初始模型f_0(x)，通常为一个常数（如目标变量均值）。

计算梯度和二阶梯度
对于当前的预测值 y^i(t)，计算每个样本的梯度 gi 和二阶梯度 hi。

构建决策树

- 计算所有特征的分裂点增益 Gain。
- 选择最大增益的特征及分裂点，进行样本划分。
- 递归构建树，直到满足停止条件（如最大深度、最小增益）。

计算叶子节点的最优权重
每个叶子节点的权重 wj计算公式：

$$w_j = -\frac{G_j}{H_j+\lambda}$$

其中 G_j 和 H_j分别是该叶子节点的梯度和二阶梯度之和。

更新预测值
计算新的预测值：

$$\hat{y}_i^{(t+1)} = \hat{y}_i^{(t)} + \eta f_t(x_i)$$

其中，η是学习率（通常取 0.1-0.3 之间）。

重复步骤 2-5
训练多个树，直到达到设定的轮数或满足提前停止条件（如验证集误差不再降低）。

5.4 XGBoost用到的参数及含义

XGBoost核心参数表

参数分类	参数名称	默认值	作用说明	典型取值范围
通用参数	`booster`	gbtree	基学习器类型（gbtree/gblinear/dart）	[gbtree, gblinear, dart]
	`nthread`	最大线程	并行计算线程数	≥1
	`random_state`	0	随机种子	任意整数
树参数	`eta` (learning_rate)	0.3	学习率/收缩系数，控制每棵树对最终结果的贡献	0.01-0.3
	`gamma`	0	节点分裂所需的最小损失减少量（正则项）	0-∞
	`max_depth`	6	树的最大深度	3-10
	`min_child_weight`	1	子节点所需的最小样本权重和（Hessian之和）	1-100
	`subsample`	1	样本采样比例	0.5-1
	`colsample_bytree`	1	每棵树特征采样比例	0.5-1
	`lambda`	1	L2正则化系数（控制叶子权重）	0-∞
	`alpha`	0	L1正则化系数（稀疏化）	0-∞
	`tree_method`	auto	树构建算法（exact, approx, hist, gpu_hist）	根据数据量选择
学习目标	`objective`	reg:square	目标函数类型（分类/回归/排序）	如 `binary:logistic`

参数分类	参数名称	默认值	作用说明	典型取值范围
	`eval_metric`	自动选择	评估指标（RMSE, MAE, logloss, AUC等）	需与目标函数匹配
	`base_score`	0.5	全局初始预测值	分类常用0.5，回归用均值
其他	`scale_pos_weight`	1	正样本权重（处理类别不平衡）	负样本数/正样本数
	`num_parallel_tree`	1	每次迭代并行生成的树数量（用于随机森林模式）	≥1
	`max_delta_step`	0	允许的单棵树最大输出值（用于逻辑回归任务中控制过拟合）	0-10
	`early_stopping_rounds`	None	早停轮数（需配合验证集使用）	10-50

关键参数深度解析

1. `eta` vs `max_depth`
    - `eta`控制整体学习速度，`max_depth`控制单棵树复杂度
    - 黄金组合：小`eta`（0.01-0.1）+ 大`max_depth`（6-10）+ 多树数量
2. 正则化三剑客
    - `gamma`：硬性分裂门槛
    - `lambda/alpha`：软性权重约束
    - `subsample/colsample`：随机性正则
3. `min_child_weight`本质
    - 对于二分类问题等价于该节点最少样本数（因Hessian=h=pred*(1-pred)）
4. `tree_method`选择策略

方法	适用场景	数据量级参考
exact	精确贪心算法（小数据）	<10^5样本

方法	适用场景	数据量级参考
hist	直方图近似算法（大数据）	>10^5样本
gpu_hist	GPU加速直方图算法	>10^6样本

调参实践建议

1. 优先级顺序
   (1) `n_estimators` + `early_stopping`
   (2) `max_depth` + `min_child_weight`
   (3) `gamma`
   (4) `subsample` + `colsample_bytree`
   (5) `eta`
2. 典型参数组合示例（二分类）

```
params = {
    'objective':'binary:logistic',
    'eval_metric':'auc',
    'eta':0.05,
    'max_depth':6,
    'subsample':0.8,
    'colsample_bytree':0.7,
    'gamma':0.1,
    'lambda':1.5,
    'early_stopping_rounds':20
}
```

5.3 个人思考与总结

- **XGBoost 主要依赖于二阶导数优化，能够更稳定地收敛**，相比于 GBDT 只使用一阶导数，XGBoost 训练更快，泛化能力更强。
- **算法在特征分裂时考虑正则化项**，这种方式有助于控制树的复杂度，避免过拟合，使得 XGBoost 具有较好的泛化能力。
- **支持并行计算和分块优化**，XGBoost 通过巧妙的计算方式（如列块数据存储、直方图优化）加速训练，适用于大规模数据集。

- 超参数调整是关键

  ，比如：

  - `max_depth` 过大会导致过拟合，通常设置在 3-10 之间；
  - `lambda` 和 `gamma` 控制正则化，防止模型过拟合；
  - `learning_rate` 过大会导致收敛不稳定，建议配合 `n_estimators` 使用。
- **适用于非线性关系的数据**，如果特征之间存在复杂的交互关系，XGBoost 往往优于线性模型（如逻辑回归），但对于高度稀疏或高维数据，可能需要特征工程来优化表现。

6.LightGBM

6.1 梯度提升树（GBT）的核心思想

梯度提升树通过迭代地构建加法模型实现优化，其预测函数为：

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F}$$

目标函数包含损失函数和正则项：

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

在每一步迭代中，通过二阶泰勒展开近似目标函数：

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

其中$g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$,$h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$。

6.2 LightGBM的核心创新

1. 直方图算法（Histogram-based）

将连续特征离散化为k个bin，建立直方图的过程：

1. 对特征值排序

2. 确定分界点：

$\{b_0, b_1, \ldots, b_k\}$,满足$b_0 = min(x), b_k = max(x)$

c. 计算每个bin的统计量：

$$G_b = \sum_{i \in I_b} g_i, H_b = \sum_{i \in I_b} h_i$$

分裂增益计算公式改进为：

$$\mathcal{G}_{split} = \frac{(\sum_{b \in B_l} G_b)^2}{\sum_{b \in B_l} H_b + \lambda} + \frac{(\sum_{b \in B_r} G_b)^2}{\sum_{b \in B_r} H_b + \lambda} - \frac{(\sum_{b \in B} G_b)^2}{\sum_{b \in B} H_b + \lambda}$$

2. 梯度单边采样（GOSS）

采样策略数学表达：

1. 按梯度绝对值降序排列
2. 保留top a%的大梯度样本集A
3. 从剩余样本中随机抽取b%的小梯度样本集B
4. 对样本集BB中的样本乘以权重系数(1−a)/b

信息增益计算调整为：

$$\tilde{\mathcal{G}}_j(d) = \frac{1}{n}[\underbrace{\sum_{x_i \in A_l} g_i + \frac{1-a}{b} \sum_{x_i \in B_l} g_i}_{\tilde{G}_l} + \underbrace{\sum_{x_i \in A_r} g_i + \frac{1-a}{b} \sum_{x_i \in B_r} g_i}_{\tilde{G}_r}]$$

3. 互斥特征捆绑（EFB）

定义特征互斥性：当特征间非零元素重叠率小于阈值$\gamma$$\gamma$时视为互斥

构建冲突图G=(V,E)，其中：

- 顶点表示特征
- 边权为特征间冲突值

通过贪心算法进行图着色：

1. 按度降序排列特征

2. 遍历特征，尝试放入现有bundle（冲突值<阈值）

3. 无法放入则创建新bundle

4. 捆绑后的特征维度从mm降为m′m'，计算复杂度从O(m)$O(m)$降为O(m′)$O(m')$

6.3 算法实现细节分析

1. 生长策略对比

策略	计算复杂度	过拟合风险	适用场景
Level-wise	O(d)	低	小数据、高噪声
Leaf-wise	O(logd)	高	大数据、低噪声

实践发现：在数据集样本量>10k时，Leaf-wise策略可提升约30%的训练速度

2.类别特征处理

LightGBM通过特殊编码方式直接处理类别特征：

1. 统计每个类别的平均目标值

2. 按统计值排序后做直方图划分

3. 最优分割点选择公式：

$$gain = \frac{(\sum_{c \in C_l} \bar{y}_c)^2}{|C_l|} + \frac{(\sum_{c \in C_r} \bar{y}_c)^2}{|C_r|}$$

6.4 工程优化

1. 内存优化：将特征值预排序后存储为uint8/uint16类型

2. 并行优化：

   - 特征并行：垂直切分特征到不同机器
   - 数据并行：水平切分数据到不同机器

3. 缓存优化：将直方图统计量存入CPU缓存行（通常64字节）

6.5 实践中的思考

1. 参数调优经验

关键参数影响分析：

参数	作用域	典型值范围	调整策略
learning_rate	全局收敛	0.01-0.3	与n_estimators联动调整
num_leaves	模型复杂度	<2^(max_depth)	从31开始逐步增加
feature_fraction	随机性控制	0.6-1.0	高维数据取较小值
lambda_l1	正则化强度	0-10	配合早停法使用

2. 常见问题处理

1. 过拟合现象：
    - 表现：训练误差<<验证误差
    - 解决方案：增加min_data_in_leaf、降低num_leaves、增大lambda_l1/l2
2. 内存不足：
    - 启用`two_round_loading`参数
    - 使用`bin_construct_sample_cnt`控制采样比例
3. 类别不平衡：
    - 设置`scale_pos_weight`参数
    - 使用平衡子采样（balanced_bootstrap=True）

【实践】

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from lightgbm import LGBMClassifier
from sklearn.metrics import accuracy_score
```

```
# 1.数据预处理
df = pd.read_csv('../data/stroke.csv')
le = LabelEncoder()
df['gender'] = le.fit_transform(df['gender'])
df['ever_married'] = le.fit_transform(df['ever_married'])
df['work_type'] = le.fit_transform(df['work_type'])
df['Residence_type'] = le.fit_transform(df['Residence_type'])
df['smoking_status'] = le.fit_transform(df['smoking_status'])
x = df.iloc[:,1:-1].values
y = df.iloc[:,-1].values
ct = ColumnTransformer(transformers=[('encoder',OneHotEncoder(),
[0,5,9])],remainder='passthrough')
x = np.array(ct.fit_transform(x))
X_train,X_test,y_train,y_test =
train_test_split(x,y,test_size=0.2,random_state=0) # 训练集测试集切分
```

```
# 2.模型构建和训练
model = LGBMClassifier()
model.fit(X_train,y_train)
y_pred = model.predict(X_test)
print("LGBM模型准确率：",accuracy_score(y_test,y_pred))

# 3.网格搜索交叉验证参数---learning_rate参数
parameters = {'learning_rate': [0.01, 0.05, 0.1]}
grid = GridSearchCV(model,
                    scoring="accuracy",
                    param_grid=parameters,
                    cv=10)
grid.fit(X_train,y_train)
print('参数learning_rate的最佳取值:{0}'.format(grid.best_params_))
print('LGBM最佳模型得分:{0}'.format(grid.best_score_))
print(grid.cv_results_['mean_test_score'])
print(grid.cv_results_['params'])

# 4.网格搜索交叉验证参数---feature_fraction参数
parameters = {'feature_fraction': [0.6, 0.8, 1],}
grid = GridSearchCV(model,
                    param_grid=parameters,
                    scoring="accuracy",
                    cv=10)
```

```
grid.fit(X_train,y_train)
print('参数feature_fraction的最佳取值:{0}'.format(grid.best_params_))
print('LGBM最佳模型得分:{0}'.format(grid.best_score_))
print(grid.cv_results_['mean_test_score'])
print(grid.cv_results_['params'])
```

运行结果：

参数feature_fraction的最佳取值:{'feature_fraction': 0.8}
LGBM最佳模型得分:0.9488769835562586
[0.9461857 0.94887698 0.9466747]
[{'feature_fraction': 0.6}, {'feature_fraction': 0.8}, {'feature_fraction': 1}]